Formal agent-oriented modeling with UML and graph transformation

Ralph Depke*, Reiko Heckel, Jochen Malte Küster

Universität Paderborn, Fachbereich 17, Warburger Str. 100, D-33098 Paderborn, Germany

Abstract

The agent paradigm can be seen as an extension of the notion of (active) objects by concepts like autonomy and cooperation. Mainstream object-oriented modeling techniques do not account for these agent-specific aspects. Therefore, dedicated techniques for agent-oriented modeling are required which are based on the concepts and notations of object-oriented modeling and extend these in order to support agent-specific concepts.

In this paper, an agent-oriented modeling technique is introduced which is based on UML notation. Graph transformation is used both on the level of modeling in order to capture agent-specific aspects and as the underlying formal semantics of the approach. Concepts of the concurrency theory of graph transformation systems following the double-pushout approach are exploited in order to formalize the relation between global requirements specification by means of sequence diagrams, and implementation-oriented design models where graph transformation rules specify the agents’ local operations. © 2002 Elsevier Science B.V. All rights reserved.

Keywords: Agent-oriented modeling; UML; Graph transformation; Formal semantics

1. Introduction

The concepts and technologies of agent-based systems become increasingly attractive to the software industry [25]. In order to benefit from advantages like increased scalability or flexibility of agent architectures, agents substitute more traditional software, or they are integrated into legacy systems. Thus, agent-oriented software development is about to become one aspect of the “normal” software development process.

Research supported by the ESPRIT Working Group APPLIGRAPH.
* Corresponding author.
E-mail addresses: depke@upb.de (R. Depke), reiko@upb.de (R. Heckel), jkuester@upb.de (J.M. Küster).
Today, most software systems are implemented in an object-oriented programming language like C++ or Java, and the analysis and design of such systems is based on object-oriented modeling languages like the UML [28]. Thus, in order to incorporate agent concepts into mainstream software development, an integrated modeling approach for object- and agent-based systems is required.

As modeling concepts, agents and objects have complementary roles: Agents act autonomously, driven by their goals and plans, thereby sensing and reacting to their environment and cooperating with other agents. Objects encapsulate data structures and operations and provide services to other objects. In this sense, Jennings et al. [35] state that “There is a fundamental mismatch between the concepts used by object-oriented developers... and the agent-oriented view”. However, the view of objects as mere service providers has its origins in the paradigms of sequential OO programming, and is no longer adequate when considering concurrent languages like Java where objects may have their own thread of control. As a modeling abstraction for concurrent objects, the concept of active object has been established [28] which has much similarity with the agent paradigm. What is still missing even in active objects is the idea of goal-driven behavior or pro-activity of agents and the related concept of autonomy. Autonomy emphasizes the fact that an agent has control over its operations: They are not called from outside like methods, but are only invoked by the agent itself in order to reach a certain goal.

Still, object-oriented modeling languages like the UML provide a good basis for the modeling of agent-based systems. In fact, a couple of approaches in the literature follow this line of reasoning proposing extensions and adaptations of object-oriented modeling techniques for agent-based systems. Iglesias et al. [23], for example, propose a methodology that covers a wide range of agent-oriented software development, from the conceptualization to the analysis and design of systems. Wooldridge et al. [35] present a methodology for agent-oriented analysis and design which is based on the FUSION method [3] and emphasizes the concept of roles. However, both approaches suffer from the usual problem of diagrammatic modeling languages: The lack of precise semantics for individual diagrams and, as a consequence, the lack of consistency rules for diagrams at different stages of the development.

It is the aim of this paper to define a modeling technique for agent-based systems building upon the concepts and notations of object-oriented modeling, in particular the UML [28]. Thereby, we pay special attention to the modeling of autonomy and cooperation of agents. We provide our technique with a formal semantics which captures the relation between different kinds of diagrams on the same level of specification as well as the consistency between different artifacts of the development activities like analysis and design.

Compared with other approaches to agent-oriented modeling, the main technical idea (which appears first in [26]) is the use of graph transformation (see, e.g., [12,13,30] for a recent collection of surveys and [1] for an introductory text) both as a modeling notation and as the underlying formal model of our technique. As a modeling notation, graph transformation rules in requirement specification and analysis allow us to capture the cooperation among several agents resulting in a joint activity. In the design activity, local graph transformation rules specify the effect of the agents’ local operations. Here,
the non-deterministic choice of the rule to be applied and the location where to apply it provide a convenient abstraction for autonomous operations. As underlying formal model of our approach, typed graph transformation systems [4] provide a natural integration of structural and dynamic aspects as well as elaborate concepts for relating systems at different levels of abstraction.

Next, in Section 2, we shall describe in more detail the properties of agent-based system we are interested in as well as the main concepts of our modeling approach. We also introduce the running example used in this paper. In the following Sections 3–5 we describe and formalize the three main activities of software modeling (i.e., requirement specification, analysis, and design) within our approach. Section 6 is concerned with the consistency between artifacts of these activities and Section 7 concludes the paper.

The paper continues previous work on agent-based systems which is documented in [6,7,8,9].

2. Agent-oriented modeling

In this section, we outline our approach to agent-oriented modeling. First, we discuss typical aspects of agent-based systems like reactivity, autonomy, pro-activity, and cooperation, and describe how these aspects are captured in our approach. Then, we survey the three main activities of system modeling, i.e., requirement specification, analysis, and design and explain how this general pattern is instantiated in our case.

Although it is difficult to find a general (technical) definition of the term agent, some important characteristics of agents can be identified which distinguish them from programs or objects [17]. Reactivity is the capability of an agent to perceive its environment and react to changes. This property can be considered as a pre-requisite for purposeful autonomy of agents, and it is already captured within the concept of active objects. In our approach, an agent’s behavior is specified by a set of transformation rules modeling its operations. Agents perceive their environment by matching the left-hand sides of their transformation rules against the current state of the system, thus searching for the occurrence of a certain pattern. Then, agents react to this occurrence by the application of a corresponding rule.

Autonomy is a property of agents that manifests itself in the non-determinism of its behavior. Non-determinism also exists for objects because in an open distributed system it is not predictable which sequence of events arrive at an object. Objects are purely reactive, i.e., they choose their operation only in dependence of the arrived events. Different to objects agents also perform internal choice, i.e., they trigger operations spontaneously. In our approach, agents possess autonomous operations that are not automatically triggered by events but may be invoked by the agents themselves. The activation of an operation demands that a corresponding situation pattern occurs in their environment (see above). If several autonomous operations are applicable in a particular situation, the decision which operation to apply is internal to the agent. The notion of autonomy for agents demands that an agent decides on its own whether an applicable
operation is indeed executed [34]. As a consequence, an agent performs autonomously internal state transitions that need not be tied with operations. The specification of such internal behavior is not in the scope of this paper. In our approach, we only restrict the behavior of agents (its operations) by statecharts which define possible execution orders of autonomous operations.

Cooperation among agents is possible if they have a common goal which is identified at run-time via negotiations. In our approach, global graph transformation rules are used in order to describe the combined effect of negotiations and the resulting joint activities of a group of agents. The communication required is specified by means of UML sequence diagrams.

An agent is supposed to be pro-active meaning that it tries to reach a certain goal. There is a close relationship between the pro-activity of an agent and its autonomy. An agent’s decision which and when to apply an autonomous operation is influenced by its drive to reach its goals. In our approach, goal-driven behavior reflects in an agent’s aim at reaching a specified goal state within the state space of the system. The specification of goals and the derivation of goal-directed behavior which, for example, may be driven by complex internal reasoning is not within the scope of this paper.

As a simple but typical example of an agent-based system we describe an online banking application where, in order to enable sophisticated services, customers may be assisted by a personal banking agent (PBA) which offers a range of advanced functionality. In particular, the PBA manages the payment of bills: When a bill is sent to the PBA by the merchant of a shop and the payment of this bill is initiated by the customer, the personal banking agent selects one of the customer’s accounts of which the bill is to be payed. This selection takes into account the transaction cost of each account which is considered.¹ Then, the amount specified in the bill is transferred from the selected account to the destination by account agents responsible for the individual accounts.

The system just described has properties that are characteristic for an agent-based system [17]: The PBA reacts to changes in its environment (like the arrival of a bill) and it modifies this environment through its actions (by paying it). It acts autonomously on behalf of the customer by selecting the account the bill is to be payed from. The agent is goal-oriented in the sense that it aims at selecting the account with the least transaction costs.

We divide the modeling process of agent-based systems in a typical sequence of activities which is already well known from the modeling of object-oriented systems. First, the requirements are specified by informal descriptions of the system’s functionality and by scenarios of important interactions. The analysis of this specification results in a model where the requirements are captured more precisely. Thereafter, in the design model the behavior that has been described globally in the analysis model is expressed by the local behavior of objects and agents.

¹ Different costs for transferring money from one account to another may result, for example, from different prices for transactions from one bank institute to another one or from a debit on the account after the money is deducted.
Within the requirements specification, in Section 3, we follow a use case-driven approach adapted to agent-based systems. Use cases representing the main external functions of the system as well as important internal interactions among agents are refined by typical scenarios which are described by means of global graph transformations and sequence diagrams. The agents’ goals are described informally as goal cases within the use case diagram.

During analysis, in Section 4, the agents and objects as well as their messages, attributes, and links, which are identified in the use cases and scenarios, are specified in an agent class diagram. The scenarios are analyzed in order to derive a more complete specification making explicit the different alternatives in the execution of a use case. The semantics of graph transformations and sequence diagrams thus shifts from optional to mandatory behavior: If the execution reaches a state satisfying a pre-condition (specified by the left-hand side of a graph transformation rule) the further interaction must follow one of the given alternatives. All diagrams of the analysis model are given a formal semantics in terms of typed graphs and graph transformation systems which, in the next step, will be extended to the design model in order to provide a notion of semantic consistency between models.

The design model in Section 5 refines the analysis model in such a way that globally described behavior is mapped to local specifications of the behavior of agents and objects. A refined class diagram introduces additional features, in particular, the signatures of the agents’ autonomous operations. The local execution order of an agent’s operations is determined by a state diagram associated to each agent class. The effect of these operations on the state of the system is described by local graph transformation rules. In the formal presentation of the design model, the states of the state diagram are encoded into the local rules so that the entire model may be represented as a typed graph transformation system.

The semantic consistency between the global graph transformation rules and sequence diagrams of the requirement specification and analysis model with the state diagrams and local transformation rules of the design model is studied in Section 6. This relation is a subject of ongoing research in the theory of object-oriented modeling. Harel et al. [20], for example, describe a method to synthesize state diagrams from requirements specified by live sequence charts (LSCs) [5]. LSCs provide an extension of message sequence charts (MSCs) [24] with lifeness constraints that allow to distinguish mandatory and optional behavior. Thus, they are able to capture the shift of interpretation between requirement specification and analysis within a single diagram.

Our notion of consistency follows the intuition of Harel et al. [20] as far as the relation between sequence diagrams and state diagrams is concerned. However, we are not dealing with the automatic transformation of MSCs into state diagrams but with the semantic consistency between the two. Moreover, in addition we take into account the specification of agent’s operations by graph transformation rules. Technically, our approach is based on a comparison of the partial order of send and receive events as specified by a sequence diagram with the causal dependencies within a graph transformation sequence. In order to extract these dependencies from a sequence, the concept of graph processes [4] is employed.
3. Requirements specification

At the beginning of a development, customers and developers have to agree on the requirements a software product has to fulfill. Therefore, at this stage of development a style of specification is appropriate which explains the functional and architectural requirements by means of informal diagrams and examples.

Use case diagrams are designed exactly for this purpose. They provide an abstract view of the system by identifying the main actors using it and the main functions that the system provides to them. In the context of agent-based systems, UML use case diagrams are extended by a special kind of actor (with square heads) representing agents. Goal cases (shown as clouds) are used in order to specify the goals of agents (cf. [23]). The use case diagram of Fig. 1, for example, identifies, besides two kinds of users, the agents PBA and Account Agent. In this way, additional architectural requirements about the distribution of the system’s functionality over different agents can be expressed. The use cases select account and pay bill that these agents participate in are internal to the system. They would not be shown in a typical UML use case diagram.

The abstract prosaic description given by use cases is illustrated by typical examples, called scenarios, of how the system behaves when a use case is performed. In the methodology of this paper, scenarios are specified in two complementary ways. The overall effect of a use case like select account is described by a pair of instance diagrams as shown in Fig. 2 modeling a before–after scenario of the use case. In the following section, this pair of diagrams shall be formally interpreted as an individual graph transformation representing the state change of objects and agents in the system.

In order to specify the communication between actors participating in a use case, UML sequence diagrams are used. In general, several such diagrams are used to capture
the behavior for a use case. They may be overlapping and need not be complete. The interaction that is necessary to select an account offering minimal transaction cost would typically be realized by the contract net protocol [16,33] which describes the negotiation between a manager and a set of potential contractors about the delegation of a task. In terms of our example, a simplified version of this protocol may be informally described as follows.

The personal banking agent solicits proposals from the account agents by issuing a call for proposals which specifies the interest in an account’s transaction costs. Account agents receiving the call for proposals are viewed as potential contractors, and are able to generate proposals to perform the task. Alternatively, account agents may refuse to propose. Once the personal banking agent receives back replies from the account agents, it evaluates the proposals and makes its choice of which account agent will perform the task. The agent of the selected proposal will be sent an acceptance message, the others will receive a notice of rejection.

A typical scenario for a personal banking agent (PBA) and two AccountAgents is depicted in Fig. 3. Other scenarios for our example would include the possibility that no AccountAgent makes a proposal or that no proposal is accepted.
4. Analysis

The rather informal and incomplete requirements specified in the first stage of the development process have to be analyzed and refined in order to serve as a basis for future design decisions. The analysis is an activity that aims at making the requirements more precise but avoids making implementation decisions. Similar to object-oriented analysis, the refined model constructed from the requirements specification is structured into (sub)models [32], a structural model, a dynamic model and a functional model. The structural model consists of an agent class diagram presenting attributes, operations and messages understood by agents. The dynamic model describes, by means of sequence diagrams the interaction among agents. The functional model specifies the overall effect of these interactions on the state of the system.

4.1. Structural model

An agent class diagram specifies the types of objects and agents, their attributes, associations, and messages. In the case of agents, messages do not automatically result in the execution of methods since agents decide autonomously how and when to react to an incoming message. The autonomous operations of agents are specified in the design model in Section 5.

Notationally, we build on class diagrams in UML [28] where agent classes are represented as active classes (with bold borders) that have an extra compartment for messages.

In the agent class diagram in Fig. 4 we have agent classes PBA and AccountAgent and object classes Bill, Account and Proposal. Associations connect the PBAs to the Bills they have to pay and AccountAgents to the Accounts they manage. Another association specifies the AccountAgents used by a PBA. A Bill specifies an amount to be paid and the Account it is to be paid to. A Proposal carries a reference to the proposer and another one to the Bill it is concerned with, and it specifies the cost for
the proposed transaction. The messages are like in the sequence diagram in Fig. 3. They are modeled in the special message compartment of the agent class.

The distinction between diagrams on the type and on the instance level is one of the most fundamental concepts in object-oriented modeling. In the context of graph transformation, this distinction is formalized by the notion of typed graphs [4]. Moreover, in order to represent data-valued attributes of objects and agents, typed attributed graphs are required.

Attributed graphs [2, 27] are graphs whose vertices or edges are colored with elements of abstract data types (like strings or natural numbers), mathematically represented as algebras over suitable signatures. For the purpose of this paper, it is enough to consider graphs with attributed vertices. Following [27], we regard attributes as edges from vertices to attribute values. Fig. 5 on the left shows an attributed type graph representing a fragment of the class diagram in Fig. 4. Classes like Bill or Account Agent are represented as vertex types while associations like pays are modeled as edge types. Messages like initPayment or propose are shown as vertex types, too, with edges pointing to recipient and parameters. Data types like int are modeled as vertex types of oval shape, and edge types pointing from a class to a data type represent attributes. E.g., edge type amount from Bill to int represents the attribute amount: int of class Bill.

An instance graph over this type graph is shown in the same figure on the right. It represents the instance diagram given by the pre-state of the transformation in Fig. 2 extended by a Bill object with amount = 5000. Formally, an instance graph over a type graph $TG$ is a graph $G$ equipped with a typing homomorphism $g : G \rightarrow TG$, i.e., a structure preserving function mapping vertices, edges, attributes, and data values $x \in G$ to their types $g(x) = t$ in $TG$. In this case, we also write $x : t \in G$. 

![Fig. 5. Attributed type and instance graph (simplified).](image-url)
4.2. Functional model

In Section 3 we have seen how the overall effect of a use case can be illustrated by a graph transformation, that is, a pair of graphs modeling a before–after scenario of the use case. Formally, this scenario can be seen as an individual test case which has to be demonstrated by the implementation of the system. However, in order to have a complete view of the use case’s overall effect, many such graph transformation pairs would be needed. Thus, a mechanism is required to specify (rather than to enumerate) pairs of graphs.

The theory of graph transformation suggests a rule-based approach to this problem. A graph transformation rule \( r = L \rightarrow R \) consists of a pair of \( TG \)-typed graphs \( L, R \) sharing the same data algebra such that the union \( L \cup R \) is defined. (This ensures that, e.g., edges which appear in both \( L \) and \( R \) are connected to the same vertices in both graphs.) The left-hand side \( L \) represents the pre-conditions of the rule while the right-hand side \( R \) describes the post-conditions. A graph transition \([15]\) from a pre-state \( G \) to a post-state \( H \), denoted by \( r(o) : G \rightarrow H \), is given by a subgraph isomorphism \( o : L \cup R \rightarrow G \cup H \), called occurrence, such that

- \( o(L) \subseteq G \) and \( o(R) \subseteq H \), i.e., the left-hand side of the rule is embedded into the pre-state and the right-hand side into the post-state
- \( o(L \setminus R) \subseteq G \setminus H \) and \( o(R \setminus L) \subseteq H \setminus G \), i.e., at least that part of \( G \) is deleted which is matched by elements of \( L \) not belonging to \( R \) and, symmetrically, at least that part of \( H \) is added which is matched by elements new in \( R \).

Notice that, during analysis, rules are considered as incomplete specifications of the transformations to be performed, i.e., additional (unspecified) changes are permitted. This (quite liberal) notion of graph transition shall be strengthened in the design model by the notion of graph transformation \( r(o) : G \Rightarrow H \) which assumes a complete specification of the changes during a step.

Fig. 6 shows three rules specifying the possible effects of the use case select account. Each rule is only concerned with the interaction of one PBA with one of its Account Agents during the execution of the contract net protocol. They specify the three possible results of each binary interaction.

The notation of graph transformation rules can also be based on UML. In [22], UML collaboration diagrams specify state transformations of graphs integrating structural and interactions aspects in the same way like graph transformation rules. Regarding this paper we dispense with the introduction of collaboration diagrams.

4.3. Dynamic model

The dynamic model complements the functional model by focusing on the communication required to execute a certain protocol. Like in the requirements specification, we use sequence diagrams to model the message flow between agents in the system. However, during analysis, we strengthen the semantics of these diagrams from an existential to an universal interpretation. This is analogous to the shift from individual transformations to universal transformation rules in the functional model. Thus, a sequence
Fig. 6. Three rules specifying the possible result of each interaction.

Fig. 7. Three sequence diagrams corresponding to the three rules in Fig. 6.

diagram associated with a graph transformation rule provides a complete specification of the interactions to be performed when the pre-condition is met.

In Fig. 7, the sequence diagrams for the banking example are presented. The first diagram models the case that the proposal of the Account Agent is accepted by the Personal Banking Agent and the second one the rejection of the proposal. The third diagram depicts the case that the Account Agent does not answer upon a call for proposal. They correspond to the rules in Fig. 6. A sequence diagram is activated when the pre-condition of the corresponding rule is met. For the rules in Fig. 6 associated with the sequence diagrams in Fig. 7, the pre-condition requires that the Account Agent is connected with the Personal Banking Agent by a uses link, and that the latter is activated by an initPayment message. Since the pre-condition is the same in all three cases, if the condition is met, the interaction between the two agents may conform to any of the three sequence diagrams.
Sequence diagrams are part of the UML, but they do not have a formal semantics in the standard. Thus, we will use sequence diagrams with the semantics of MSCs [24]. This is straightforward as sequence diagrams have originally evolved from MSCs (and there are even attempts to unify them again [31]). Thus, a sequence diagram is seen as a specification of a partial order over the set of events representing the sending and receiving of messages. These events are visible in the diagram as the sources and targets of message arrows at the vertical instance lines. As defined in the standard [24], all events along an instance line are ordered, as are any pairs of events signaling the sending and receiving of the same message.

Consider as an example the left sequence diagram in Fig. 7 and the associated first transformation rule in Fig. 6, jointly modeling a successful negotiation between a PBA and an Account Agent. Fig. 8 shows the core graph of this interaction, i.e., a graph representing all objects, links, and messages occurring in the rule and the sequence diagram. (We use the representation of messages as vertices introduced in the structural model.) The set of events of this interaction is now given as

\[ \{ \text{rec}(m_1), \ldots, \text{rec}(m_4), \text{snd}(m_2), \ldots, \text{snd}(m_4) \} \]

with \( \text{snd}(m_i) \) and \( \text{rec}(m_i) \) representing, respectively, the event of sending and receiving message \( m_i \). The partial order of events is generated by the orderings induced by the two vertical instance lines

\[ \text{rec}(m_1) \leq \text{snd}(m_2) \leq \text{rec}(m_3) \leq \text{snd}(m_4) \text{ and } \text{rec}(m_2) \leq \text{snd}(m_3) \leq \text{rec}(m_4) \]

and the causal dependency between the submission and reception of the same message \( \text{snd}(m_i) \leq \text{rec}(m_i) \) for \( i \in \{2, 3, 4\} \). The interaction is thus given by the triple \( I = \langle G, H, E, \leq \rangle \) where \( \langle E, \leq \rangle \) is the partial order of events over \( C = G \cup H \).
5. Design

The analysis activity is concerned with developing a model of what the system is supposed to do. The design model elaborates the analysis model concentrating on the question how the system will function. As a consequence, the focus of models is shifted from a global view on the system during analysis to a local view, thus providing the basis for an implementation.

Similar to the analysis model, our design model consists of a structural model, a dynamic model and a functional model. The structural model consists of a refined agent class diagram presenting attributes, operations and messages understood by agents. The dynamic model describes, by means of state diagrams for each class, the order in which operations of classes can be performed. The functional model specifies the effect of operations using graph transformation rules.

Models constructed during design are refinements of models of the requirements specification and of the analysis. As a consequence, the design model must be syntactically and semantically consistent with the earlier model. Using concepts of the graph transformation theory, in this and the next section, these consistency rules are formally expressed.

5.1. Structural model

The class diagram of the design refines the class diagram of the analysis adding, in particular, the signatures of the agent’s autonomous operations for which an extra compartment is provided. Notice the difference with methods as specified in the method compartment of objects: An agent’s operations are autonomous, that is, they are never called by another object or agent but only executed under control of the agent itself (cf. Section 2). As a consequence, we distinguish agent’s messages and operations while in the case of objects, both notions are integrated in the notion of a method.

Formally, the signatures of both operations and methods are represented by a family of sets of operation symbols $OP = (OP_w)_{w \in TG^+}$, indexed by non-empty lists of types of the type graph $TG$ representing the class diagram. For $w = t_0 \ldots t_n$, the first argument $v_0$ is the type representing the class where the method is defined. A pair $(TG, OP)$ of a type graph and a corresponding operation signature is also called a transformation signature.

Consider, for example, the class diagram in Fig. 9. The AccountAgent has two autonomous operations, answerCFP and doTransaction both belonging to the set $OP_{AccountAgent,Bill}$. These two operations reflect the agent’s autonomy. The agent decides by itself whether and when to answer a call for proposal and to perform a transaction on the account it manages.

Besides operations, the design level class diagram may also add other model elements like classes, associations, attributes, and messages. In the banking example, several new associations are introduced, e.g., a sent association between Bill and AccountAgent for expressing that the request for proposals concerning this bill has already been sent to this agent. Syntactic consistency between the class diagrams at the analysis level and at the design level can be formally expressed by a signature morphism.
A morphism \( h: \langle TG, OP \rangle \rightarrow \langle TG', OP' \rangle \) between two transformation signatures consists of a graph homomorphism \( h_{TG} \) between the type graphs and a \( TG^+ \)-indexed family of mappings of operation symbols \( (h_w: OP \rightarrow OP'_w)_{w \in TG^+} \) such that \( op: t_1 \ldots t_n \) implies that \( h_w(m): h_{TG}(t_1) \ldots h_{TG}(t_n) \). Thus, signature morphisms allow the extension, renaming and homomorphic transformation of class diagrams, preserving the structure of classes and associations, and the typing of methods.

The benefit of making explicit the relation between the class diagrams is that rules and diagrams can automatically be translated along these relationships. This is essential for defining the semantic consistency between analysis and design in Section 6. In particular, a homomorphism \( h_{TG}: TG \rightarrow TG' \) between type graphs induces a retyping of \( TG \)-typed instance graphs to \( TG' \)-typed ones. For a \( TG \)-typed instance graph \( G \) with typing \( g: G \rightarrow TG \), the retyping yields the same instance graph typed by \( h_{TG} \circ g: G \rightarrow TG' \). Thus, the retyped graph has the same elements and connections like the original one, but the types of these elements may be renamed, formerly different types may be identified with each other, and new types may be introduced. The retyping operation extends easily to occurrences and transformations. On this background, in Section 6 we may assume that all diagrams, rules, graphs, transformations, etc., which are dependent on a class diagram or type graph, are retyped appropriately to the class diagram of the design model.

5.2. Dynamic model

By a state diagram for each agent class, the dynamic model specifies the ordering of operations an agent of this class may perform. As our agents are autonomous, they do
not react to events of their environment immediately with a specific action but rather decide themselves when and how to react. As a consequence, transitions are not labeled with an event and an action but only with the name of the operation. In particular, our usage of statecharts is semantically different from traditional approaches [19]: The left-hand side of a graph transformation rule, i.e., an operation, forms a condition regarding the current state of an agent’s environment which has to be matched in order to apply the rule. An event is represented by a message which is also contained in the left-hand side of the rule. Thus, an operation is triggered by the coincidence of a fulfilled condition and the arrival of a message event. Possibly, an operation can also be triggered without events if there is no message within the left-hand side of the rule. This corresponds well with the autonomy of agents. In traditional statecharts a trigger is always activated by the occurrence of an event which results in pure reactivity. The notion of statechart introduced by Reggio et al. [29] comes closest to our understanding.

The characteristics of agents also show in the non-determinism inherent to this usage of statecharts. Consider, as an example, the statechart for the AccountAgent in Fig. 10. From the first state, this agent may either proceed to the proposed state by answering a call for proposal or it may decide not to propose and proceed to the final state. This internal decision is not specified at this level of abstraction.

5.3. Functional model

In the functional model, the operations declared in the structural model are specified by typed graph transformation rules. Whereas the dynamic model is concerned with the order of operations the functional model shows how operations change the state of the system. Each operation has a pre-condition depicted on the left-hand side of the graph
transformation rule and the result of the operation depicted on the right-hand side. An agent’s operations only affect that part of the system state which the agent can access locally. Therefore, objects can only be modified if in the state they are reached by a path of links originating from the agent. Other agents are influenced by messages sent to them.

The first operation `getBill` (see Fig. 10) triggers the agent to issue requests for proposals for a given bill. In Fig. 11, the operation `sendCFP` of the PBA is specified. If a PBA has not yet sent a call to a particular AccountAgent (expressed by the negative context condition for the sent link) the PBA may use the second rule for issuing the call to this agent.

On reception of a call for proposal message, an AccountAgent may decide to propose to the PBA by sending a Proposal with the costs for the required transaction for paying the bill as specified by the rule `answerCFP` in Fig. 12. The alternative rule for `ignoreProp` is not shown. It has the same pre-condition and the only effect of removing the cfp message.

Receiving a proposal, the PBA may either reject it if it has bigger cost than the best proposal received so far or it may record this proposal as its current favorite.
Rejection results in the sending of a reject message. The performing rule `recordProp` is not shown. If the new proposal is better than the one selected so far, it is selected as new favorite by the rule `recordProp` (see Fig. 13) and the sender of the old favorite proposal is sent a reject message. The first proposal is recorded when the agent stops sending calls which is specified by the rule `stopCFP`, see Fig. 14.

Whenever the PBA has received enough proposals it decides to accept the current best by sending an accept message. Upon reception of this message, the AccountAgent records its proposal as accepted. When rejected, the agent deletes its proposal. The rules for the operations `acceptProp`, `getAccepted` and `getRejected` are not shown.

With respect to the autonomy of agents, we explicitly allow that two graph transformation rules may have the same left-hand side. It is the decision of the agent itself which operation to perform, on the basis to reach its goals and possibly on its internal representation. Currently, in the design model we abstract from these aspects.

The integration of the functional and the dynamic model is achieved by encoding the states of the state diagram into the left- and right-hand sides of the rules. For this purpose we introduce, for each state, a new vertex type together with an edge type from the state type to the agent type the state diagram is associated with. Thus, in our example, new vertex types `waiting`, `sending` and `receiving` are introduced along with edge types to the vertex type PBA. Moreover, for a rule specifying an operation
op labeling a transition from state $A$ to state $B$ in a state diagram, we add an $A$-vertex in the left-hand side and a $B$-vertex in the right-hand side, both connected to the self-agent of the operation. In this way, a rule can only be applied if the agent is in the state displayed in the pre-condition, and after application the agent is in the state displayed in the result. In Fig. 14, the corresponding formal presentation of the operation stopCFP is shown by the two gray shaded vertex instances of type waiting and receiving. Additionally, the formal representation of a message of type propose is depicted by a vertex along with edges to the recipient, the sender and a parameter.

Due to the encoding of state diagrams into the rules, the design model can be formally represented as a graph transformation system. A graph transformation system $\mathcal{G} = \langle TG, OP, R \rangle$ consists of a type graph $TG$, an operation signature $OP$, and a set of graph transformation rules $R$ equipped with the names and formal parameters of the operations which are specified (cf. the examples in Figs. 11–14). In the following section, this formal presentation shall be used in order to define the semantic consistency of the design model with the analysis model and the requirement specification.

6. Semantic consistency

The syntactic consistency of the class diagrams of the analysis and design models has been expressed by a structure-preserving mapping between the corresponding type graphs and operation signatures. Based on the retyping of diagrams of earlier models to diagrams over the class diagram of the design model (cf. discussion on page 15), we shall now study the semantic consistency between the models.

In the analysis model, the pre- and post-conditions as well as the communication associated with a use case have been described by global graph transformation rules and sequence diagrams, respectively. Our design model must conform to these specifications: Whenever the pre-condition of a global graph transformation rule is satisfied in a state conforming to the design class diagram, a corresponding sequence of
transformations using the local rules of the design model must exist which implements (at least) the same overall effect. Moreover, the sequence must realize the message flow specified in this case by the corresponding sequence diagram. Next, we give an operational semantics to the design model which allows us to make more precise these requirements.

First, a notion of rule application is defined which reflects the intuition of the design model that the transformations on instance graphs are completely specified by the rules. A graph transformation from a pre-state $G$ to a post-state $H$, denoted by $r(o): G \Rightarrow H$, is given by a subgraph isomorphism $o: L \cup R \rightarrow G \cup H$, called occurrence, such that

- $o(L) \subseteq G$ and $o(R) \subseteq H$, i.e., the left-hand side of the rule is embedded into the pre-state and the right-hand side into the post-state,
- $o(L \setminus R) = G \setminus H$ and $o(R \setminus L) = H \setminus G$, i.e., exactly that part of $G$ is deleted which is matched by elements of $L$ not belonging to $R$ and, symmetrically, exactly that part of $H$ is added which is matched by elements new in $R$.

Therefore, a graph transformation $r(o): G \Rightarrow H$ is a graph transition $r(o): G \rightarrow H$ where, in addition, $o(L \setminus R) \supseteq G \setminus H$ and $o(R \setminus L) \supseteq H \setminus G$ (cf. Section 4).

A sequential model of the computations in a graph transformation system $\mathcal{G}$ is defined by the set of all sequences of transformation steps in $\mathcal{G}$. We assume to be given a set $S$ of $TG$-typed graphs as states, such that their union $\bar{S} = \bigcup_{G \in S} G$ is well-defined. A trace $\tau = \tau_1 \ldots \tau_n: G \Rightarrow H$ in $\mathcal{G}$ with $G, H \in S$ is a sequence of transformations $\tau_i = r_i(o_i): G_i \Rightarrow H_i$ with $G_1 = G$, $H_n = H$ and $G_i, H_i \in \bar{S}$ for all $1 \leq i \leq n$, and such that $H_i = G_{i+1}$ for all $1 \leq i < n$.

We denote a trace by a sequence of expressions $op(P_1, \ldots, P_n)$ consisting of an operation $op$ identifying the rule with actual parameters $P_1, \ldots, P_n$ specifying the occurrence. The following trace realizes the before–after scenario of Fig. 2

\[
\begin{align*}
&\text{A.getBill(B); } \text{A.sendCFP(B, Acc1); } \text{A.sendCFP(B, Acc2);} \\
&\text{Acc2.answerCFP(B); } \text{Acc1.answerCFP(B); } \text{A.stopCFP(P2); } \text{A.recordProp(P1); } \\
&\text{A.acceptProp(P1); } \text{A.rejectProp(P2); } \text{Acc2.getRejected(P2); } \text{Acc1.getAccepted(P1).}
\end{align*}
\]

More formally, we say that a trace $\tau: G \Rightarrow H$ realizes a global transformation rule $r = L \rightarrow R$ (of the requirement specification or the analysis model) if there exists an occurrence $o: L \cup R \rightarrow G \cup H$ such that $r(o): G \rightarrow H$ forms a graph transition. That means, the trace implements at least the same overall effect as specified by the rule $r$.

If a trace is to be seen as the implementation of an interaction (like the negotiation between two agents) we have to ensure that, besides pre- and post-conditions specified by the global rule, it respects also the message order described by the corresponding sequence diagram. In order to be able to compare traces and sequence diagrams, we need to understand the different nature of the two representations. First, the sequence diagram is only concerned with a local view of the system, i.e., the agents participating in the specified interaction. A trace, on the other hand, is a global sequence

\[\begin{align*}
\text{A.getBill(B); } \text{A.sendCFP(B, Acc1); } \text{A.sendCFP(B, Acc2);} \\
\text{Acc2.answerCFP(B); } \text{Acc1.answerCFP(B); } \text{A.stopCFP(P2); } \text{A.recordProp(P1); } \\
\text{A.acceptProp(P1); } \text{A.rejectProp(P2); } \text{Acc2.getRejected(P2); } \text{Acc1.getAccepted(P1).}
\end{align*}\]

\[\begin{align*}
\text{A.getBill(B); } \text{A.sendCFP(B, Acc1); } \text{A.sendCFP(B, Acc2);} \\
\text{Acc2.answerCFP(B); } \text{Acc1.answerCFP(B); } \text{A.stopCFP(P2); } \text{A.recordProp(P1); } \\
\text{A.acceptProp(P1); } \text{A.rejectProp(P2); } \text{Acc2.getRejected(P2); } \text{Acc1.getAccepted(P1).}
\end{align*}\]

\[\begin{align*}
\text{A.getBill(B); } \text{A.sendCFP(B, Acc1); } \text{A.sendCFP(B, Acc2);} \\
\text{Acc2.answerCFP(B); } \text{Acc1.answerCFP(B); } \text{A.stopCFP(P2); } \text{A.recordProp(P1); } \\
\text{A.acceptProp(P1); } \text{A.rejectProp(P2); } \text{Acc2.getRejected(P2); } \text{Acc1.getAccepted(P1).}
\end{align*}\]

\[\begin{align*}
\text{A.getBill(B); } \text{A.sendCFP(B, Acc1); } \text{A.sendCFP(B, Acc2);} \\
\text{Acc2.answerCFP(B); } \text{Acc1.answerCFP(B); } \text{A.stopCFP(P2); } \text{A.recordProp(P1); } \\
\text{A.acceptProp(P1); } \text{A.rejectProp(P2); } \text{Acc2.getRejected(P2); } \text{Acc1.getAccepted(P1).}
\end{align*}\]

\[\begin{align*}
\text{A.getBill(B); } \text{A.sendCFP(B, Acc1); } \text{A.sendCFP(B, Acc2);} \\
\text{Acc2.answerCFP(B); } \text{Acc1.answerCFP(B); } \text{A.stopCFP(P2); } \text{A.recordProp(P1); } \\
\text{A.acceptProp(P1); } \text{A.rejectProp(P2); } \text{Acc2.getRejected(P2); } \text{Acc1.getAccepted(P1).}
\end{align*}\]
where operations of different interactions are interleaved. Second, the order of operations (events) specified in a sequence diagram is in general partial while a sequence represents a total (linear) order of operations.

In order to resolve this mismatch, we abstract from the concrete ordering of steps in a sequence. By the structure obtained in this way, called graph process \[4\], only the data dependencies between the steps are recorded using the occurrences of the rules, while the information about the scheduling of independent steps is lost. Formally, a graph process \(\rho\) in a graph transformation system \(G\) is a set of transformation steps in \(G\) such that there exists a linear order on \(\rho\) turning the set into a trace.

The causal dependencies between the steps in a process are represented by a partial order which can be derived from a process \(\rho\) as follows. Let \(C \subseteq \mathcal{S}\) be the union of all pre- and post-graphs of the transformations in \(\rho\). Let \(\rho(o) : G \Rightarrow H\) be one arbitrary transformation in \(\rho\), \(r = L \rightarrow R\) be the corresponding rule, and \(e\) be any arc, node, or attribute in \(C\). We say that

- \(\rho(o)\) consumes \(e\) if \(e \in o(L \setminus R)\),
- \(\rho(o)\) creates \(e\) if \(e \in o(R \setminus L)\),
- \(\rho(o)\) preserves \(e\) if \(e \in o(R \cap L)\).

The partial order \(\preceq\) defined on \(\rho \cup C\), called causal relation \[4\], is defined as the transitive and reflexive closure of the relation \(<\) where

- \(e < r_1(o_1)\) if \(r_1(o_1)\) consumes \(e\),
- \(r_1(o_1) < e\) if \(r_1(o_1)\) creates \(e\),
- \(r_1(o_1) < r_2(o_2)\) if \(r_1(o_1)\) creates \(e\) and \(r_2(o_2)\) preserves \(e\), or \(r_1(o_1)\) preserves \(e\) and \(r_2(o_2)\) consumes \(e\).

The causal order can be used in order to recover from a process \(\rho\) the initial and the final graphs of a trace, formed by the sets of minimal and maximal elements of \(C\), respectively. This justifies to denote a process as \(\rho/C : G \Rightarrow H\) where graphs \(C, G\) and \(H\) are determined by the set of transformations \(\rho\).

Based on the partial order induced by a process, we can now formalize the idea of realization of a sequence diagram. Consider a trace \(\tau : G \Rightarrow H\) which realizes a global transformation rule \(r = L \rightarrow R\) via an occurrence \(o : L \cup R \rightarrow G \cup H\). We say that \(\tau\) realizes the sequence diagram associated with \(r\) if there exists a mapping \(rl : E \rightarrow \rho\) from the set of events of the sequence diagram to the set of steps forming the process such that \(rl\) preserves the partial order of events, that is, \(rl(e_1) \preceq rl(e_2)\) for all \(e_1 \leq e_2 \in E\). We also require that the mapping \(rl\) is compatible with the occurrence \(o\) in the sense that \(o\) extends from \(L \cup R\) to the core graph \(C\) of the diagram (that is, to the messages) such that \(rl(rec(m))\) consumes \(o(m)\) and \(rl(snd(m))\) creates \(o(m)\) for all messages \(m\).

The following mapping \(rl : E \rightarrow \rho\) shows that the trace above realizes the first sequence diagram of Fig. 7. It associates events \(snd(m)\) and \(rec(m)\) to steps in the trace.

\[
\begin{align*}
rec(m_1) & \mapsto A.getBill(B) \\
snd(m_2) & \mapsto A.sndCFP(B, Acc1) \\
rec(m_3) & \mapsto A.recordProp(P1) \\
snd(m_4) & \mapsto A.acceptProp(P1)
\end{align*}
\]
This mapping is indeed a realization because the partial order of events specified in the sequence diagram is preserved. For example, the dependency $\text{snd}(m_2) \leq \text{rec}(m_2)$ is preserved because $\text{rl}((\text{snd}(m_2)) = \text{A}.\text{sndCFP}(\text{B}, \text{Acc1})$ creates the cfp message and $\text{rl}((\text{rec}(m_2)) = \text{Acc1}.\text{answerProp}(\text{B})$ consumes this message. Thus, according to the definition of the partial order above, $\text{A}.\text{sndCFP}(\text{B}, \text{Acc1}) \leq \text{Acc1}.\text{answerProp}(\text{B})$. The dependency $\text{rec}(m_1) \leq \text{snd}(m_2)$ is realized in the trace by means of the encoding of the sending state into the right-hand side of rule $\text{A}.\text{getBill}(\text{B})$ and the left-hand side of rule $\text{A}.\text{sndCFP}(\text{B}, \text{Acc1})$ (cf. rule $\text{stopCFP}$ in its visual and formal presentation in 14, respectively).

According to the different interpretation of rules and sequence diagrams in requirement specification and analysis, we define two notions of consistency. In the first case, we assume to be given typical scenarios for each use case. Thus, we require that each of these scenarios has at least one realization in the design model. In the second case, a set of alternative rules and sequence diagrams delivers a universal specification of the transformation and communication associated with a use case. Therefore, we ask that from each state graph in the design model satisfying the pre-condition of the use case, there exists a realization of at least one of the alternatives.

- For each rule $r$ (and corresponding sequence diagram) in the requirement specification, the design model is consistent with $r$ if there exists a trace $\tau: G \Rightarrow H$ which realizes $r$.
- For each set $R$ of alternative rules $r_i = L \rightarrow R_i$ (and corresponding sequence diagrams) in the analysis model and for every occurrence $o_L: L \rightarrow G$ in a graph $G$ conforming to the design class diagram, there exists a rule $r_i$ in $R$ and a trace $\tau: G \Rightarrow H$ which realizes $r_i$ such that the occurrence $o: L \cup R \rightarrow G \cup H$ of the corresponding transition is an extension of $o_L$ (that is, $o_L = o|_L$).

The partial-order-based approach to relate sequence diagrams and global system traces is not the only possible solution to this problem. Other approaches rely on the construction of a global system automaton [20] which is obtained by computing the product of the state diagrams of the design model. Traces of this automaton can then be compared with interleavings of the partial order obtained from the sequence diagram or MSC.

In Fig. 15, the product automaton of the state diagrams of agent classes PBA and AccountAgent from Fig. 10 is constructed. The example shows that the automata-theoretic approach does not take into account data dependencies defined by the functional model because the effect of operations on the state of the system is not specified in the state diagram. For example, the $\text{acc}.\text{answerCFP}$ rule of an AccountAgent can only be applied if there has already been issued a cfp message from the PBA. As a consequence, many transitions in the global system automaton are never possible if the pre-conditions of the operations are taken into account. In Fig. 15, only solid arcs describe possible transitions of the product automaton. Dotted arcs are transitions that result from the product of the state diagrams but they are impossible because data is missing in the source state of the transition in order to apply a graph transformation rule. For example, the operation (rule) $\text{acc}.\text{answerCFP}$ cannot be applied in the state (1,5) because no cfp message has been issued before. Another disadvantage is that all possible interleavings are explicitly visible as alternative paths within the automaton. In
Fig. 15 for example, two possible interleavings of the operations acc.answerCFP and a.stopCFP are given by the paths between the states (2,5) and (3,6) via states (2,6) and (3,5). In comparison with this, a single partial order represents a whole equivalence class of global traces, i.e., all its interleavings.

7. Conclusion

In this paper, we have presented an approach to agent-oriented modeling based on UML notation and concepts of typed graph transformation systems [4]. Extending the notion of active object from object-oriented modeling, specific support is provided for characteristic aspects of agent-based systems like autonomy and cooperation of agents.

The theory of graph transformation also provides the mathematical background for the formalization of the approach. In particular, the theory of graph processes [4] and concepts of refinement of graph transformation systems [15,18,21] are used in order to formalize the consistency between requirement specification, analysis and design in agent-oriented modeling.

Based on the work of this article, we developed a concept of roles that enhances the transition between different stages of the agent-oriented modeling process in a systematic way [10]. Roles are used for modeling participants of protocols where each
agent participating in a protocol is assigned a certain role which it plays as long as the protocol is performed. Roles are introduced during requirement specification and in the analysis they are assigned to protocols. In the design model the protocols are expressed by local operations of the agents’ roles. In [11], we modified the DPO-semantics of graph transformation rules for a formal treatment of role properties. By use of a concept of rules on the meta level that generate graph transformations from graph transformation rules the semantics of graph transformation rules containing roles can be changed appropriately.

References


